Microtasks: A Platform for Task Centered Collaboration

Mohit Gupta, Joseph Joy, Krishna Mehra, Gopal Srinivasa

Microsoft Research India

ABSTRACT

We consider the class of applications that involve users working asynchronously or synchronously in small groups to complete a set of tasks. Despite wide applicability, broad adoption of these applications remains largely unrealized, due to the high cost of development and deployment. We introduce an abstraction for these applications that enables a substantial portion of application complexity to be factored out into a common infrastructure we call the Microtasks Environment. We describe the environment and its simplified programming model, and share our experiences in developing and deploying applications that are in actual use.

Author Keywords

Collaboration platform, human-centered computing.

INTRODUCTION

The Internet has enabled a class of applications where people collaborate towards completing tasks, having a variety of motivations to participate and without necessarily knowing each other. For example, the Distributed Proofreaders (DP) project[1] enables volunteers to sign up to help with correction, formatting and proofreading tasks associated with getting books online as part of Project Gutenberg[2]. Other applications go beyond farming out pieces of work to individuals working independently, leveraging instead the *synchronous interactions* among people. For example, the ESP Game[3] is a 2-person synchronous image tagging game that cleverly leverages competiveness to both motivate people to both contribute, and help cross-verify contributions., The motivation for the Internet community to participate can be monetary, as is the case with Amazon's Mechanical Turk (MTurk) service [4], and Microsoft's Task Market [5], or it can be entertainment, as is the case with the ESP game and the other games on the "Games with a purpose" portal [6][7]. At other times, users have worked on such tasks motivated by altruistic goals, as the search for Jim Gray demonstrates [8]. Some companies are using such games for business functions like improving product quality [9], while others have incorporated such applications into their business plans.[10][11].

By leveraging interactions, the Internet becomes a source of collective, cross-validated wisdom that can be tapped to create, transform, and annotate digital information [12].

Today, an individual without significant engineering resources has limited options to tap into this pool of human resources, either within an organization or across the Internet, in the manner the DP project or the ESP game does. While the MTurk service [4] supports a fixed set of single-user task templates, it does not natively support tasks with customized orchestrations (as in DP) or tasks requiring synchronous collaboration among multiple parties (as in ESP). MTurk also requires a separate web service to be setup for tasks with nontrivial UIs.

One could build a distributed application or service, but it takes significant engineering resources to build and deploy such applications due to the need for user management, data management, concurrency, messaging, application health monitoring and other reasons we elaborate on in later sections. This is in stark contrast to standalone applications: script-driven desktop applications, high-level languages and frameworks have empowered people from broad range of disciplines, including non-IT disciplines, to either build a standalone application from scratch or to customize existing applications for specific needs.

The main goal of the work described in this paper is to significantly lower the technological entry barrier for the development and deployment of a broad class of collaborative applications, which we call Task Centered Collaboration applications, defined below:

> **Task Centered Collaboration** (TCC) applications facilitate the processing of a large number of work items, or **Tasks**, by people working independently or in small, synchronously collaborating groups.

A second, longer term goal of our work is to build a platform for the study of computer mediated communication and human-computer interactions for this class of applications.

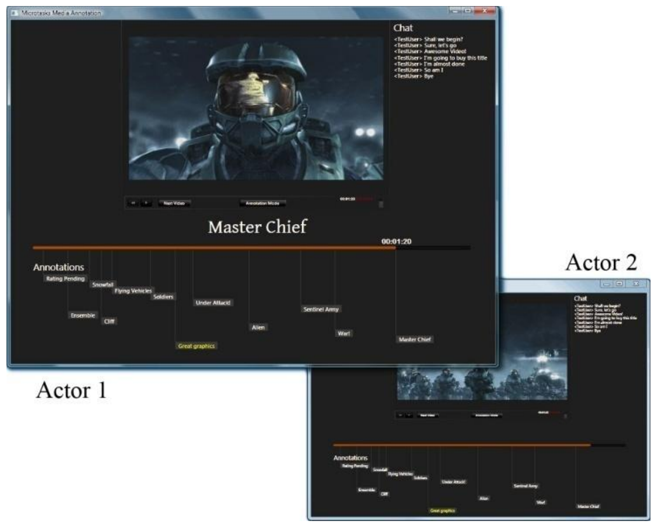
In this paper, we describe a system, called the **Microtasks Environment**, for the development and deployment of TCC applications. To our knowledge, this is the first system that enables ease of development and deployment, combined with extensibility, for this class of applications.

The key contributions of our work are:

- A vocabulary that enables application authors to clearly specify and design TCC applications.
- An abstraction, applicable to the broad class of TCC applications, which enables the factoring out of

significant portions of the application into common infrastructure.
- A programming model that enables a developer to focus on the UI and logic associated with operations on a single task, without being exposed to issues such as security, authentication, synchronization, concurrency, persisting data, and binding users to activities. The model includes a serialized, typed message passing model to facilitate synchronous collaboration when working on a task.
- An implementation of the system on which we have developed and deployed a diverse set of applications that include: tagging portions of images with text, multilingual annotation of geographic points of interests and collaborative video annotation. This system is in actual use and is providing value to ongoing projects in the area of video annotation for education, multilingual corpora collection, and a machine learning research project [13].

Figure 1 shows a screenshot of SMA, a collaborative media annotation application built and deployed using our system. SMA enables a pool of (potentially geographically distributed) volunteers to collaboratively annotate existing video or audio content. Each media clip is worked on by a transient, ad-hoc group, automatically put together by the Microtasks Environment, and consisting (typically) of one to three users, who contribute timeline annotations while reviewing the clip. Contributed annotations are shared in real time, and the collective annotated results are check-pointed to persistent storage maintained by the Microtasks Environment. The application includes a group chat window that facilitates discussions amongst the group. SMA, which is in actual use for the annotation of educational videos, is used as an example throughout this paper to illustrate the various concepts and the programming model we introduce.



<TestUser> Shall we begin?
<TestUser> Sure, let's go
<TestUser> Awesome Video!
<TestUser> I'm going to buy this title
<TestUser> I'm almost done
<TestUser> So am I
<TestUser> Bye

Figure 1: Two instances of SMA, a collaborative media annotation application

CHARACTERIZING TCC APPLICATIONS

A unifying characteristic of the class of collaborative applications we consider is that collaboration revolves around independent chunks of data that are operated upon by the users of these applications. During the course of one or more user interactions, the data chunks get transformed or annotated with additional data. We call this data chunk that changes state and accumulates content, a **Task.** Note that our definition of Task is focused on the *data*, not the user interaction activities that may operate on it. We use this definition consistently.

Some examples of applications that operate on Tasks are:
1. Surveys (the simplest form of a TCC application).
2. Crowd-sourcing applications at various scales, ranging from sending out information to be annotated by a targeted group of people, to large scale tagging applications such as HotOrNot [14]. Tags can also be collected asynchronously over time, with each user adding new tags to the images that provide sufficient permissions as in the case of *Flickr* [15].
3. Synchronous collaboration by small sets of people on small, data-driven shared tasks (such as the SMA media annotation application). This includes one- and two-person games with data collection intent such as the ESP Game [3].

Tasks can be operated on in multiple stages, and involve some amount of workflow, such as distributed proofreading (see for example [1]). Another example of this is seen in Peek-a-boom [16] where two users synchronously collaborate to annotate sections of an image, based on tags earlier submitted in the ESP game.

Building TCC applications is not easy despite the availability of a plethora of frameworks. This is not only due to the domain-specific functionality required by TCC applications (for instance, matching users with certain qualifications to appropriate tasks) but also the need to synchronize the modifications to shared data by multiple people with varying roles, capabilities and trust levels. The authors have built several TCC applications from ground up, and can speak from firsthand experience. The difficulty of building collaborative, distributed applications is also highlighted in [17].

One can lower the barrier to the development and deployment of TCC applications by sufficiently constraining the problem and this is effectively done in a number of cases. For example MTurk enables non-technical users to create template-based Human Intelligence Tasks (HITs) for some common scenarios such as editing documents or image tagging. However, adding the slightest sophistication to the interaction such as the marking-out parts of an image demands one to create a complete web application (as in GIS Image Tagging App [18] on MTurk). Even lightweight applications on the Internet require components to be well engineered before they can be published – these components include authentication, data storage, security, and monitoring. Using web application frameworks such as Ruby on Rails, ASP.NET and others reduces this burden, but they in turn require significant technical knowledge.

When synchronous *collaboration* is involved (as in games), the situation gets considerably more complex, as there would need to be communication and synchronization amongst the users working on the same Task. The developer would need to be well versed with distributed programming techniques. The dynamic mapping of specific Tasks and user interaction activities to users, given application-specific constraints, is also non-trivial. Scalability and robustness also need to be addressed, given that a large number of Tasks (especially if they involve lightweight interactions) can be in execution simultaneously. The computer gaming community has worked on various abstractions [19], but there is a significant learning curve and the environment is highly specialized to the gaming experience.

Deploying such applications is non-trivial and there are significant operational and administrative overheads. For an application that works at Internet scale, one may have to interface with replicated databases and clusters of servers, all of which requires specialized technical knowledge.

THE CASE FOR A COMMON INFRASTRUCTURE

Most of the TCC application examples in the previous section are fairly large and successful Internet applications. We believe that the opportunity for such applications *within* an organization (or informal workgroup) is huge. Every instance of work being partitioned amongst people and eventually collated is an opportunity for a TCC application. Some examples are listed below:

1. A User Interface designer wanting to send out versions of her UI for A-B testing.
2. A teacher wanting to compose exercises in the form of casual games, as motivated by the MIT Games-to-Teach project [20].
3. An individual may want to get her media file tagging and annotating "outsourced."

However, we believe the potential is largely untapped due to the difficulty of building and deploying these applications.

Despite the varied interaction modalities of a user (or users) working on a specific task there are many commonalities in these applications that can be factored out into a single infrastructure, such as:

- User authentication and management
- Database management and access
- Synchronization for multi-user collaborative tasks
- Managing task lifecycles
- Monitoring and evaluation framework
- Administrative interface for managing Tasks and analyzing usage statistics.
- Security and protection from malicious user behavior.

Our goal has been to move beyond building a set of common libraries and sophisticated design guidelines (thereby leaving the application author to compose and deploy the application), and get to a world where the author's burden is greatly minimized, so that she can focus on the specifics of the application (which is primarily the details of how users interact with a Task in the context of specific User Interaction activities, as well as the ability to upload and download aggregate data).

This goal has led us to compose an abstraction that encompasses the class of TCC applications, and the creation of the Microtasks Environment, described in the next section.

THE MICROTASKS ENVIRONMENT

The Microtasks Environment is guided by the following design goals:

1. Present a simple programming model to the application author that supports the full class of TCC applications. All concepts and components that are not specific to the particular application should be factored out into a common infrastructure.
2. Provide a hosting environment for these applications.
3. Provide multiple points of extensibility. For example, the application author should have control over the user interface, and should be able to run arbitrary "business intelligence" logic.
4. Provide persistent storage. Persisted data should be query-able by the business logic.
5. Support complex Task workflows.
6. Ensure that the choice of platform for the end-client interaction component is not constrained by the framework. Common platforms could include desktop applications, web browsers and popular social networking platforms.
7. Support an evaluation framework that authors can tap into to introspect on the running of the application.

Overview

The Microtasks Environment implements the Microtasks abstraction, which is the model of the system presented to the application author (**Author**). The semantics and terminology of the entities in the abstraction have been designed to closely follow the informal characterization of the full swathe of TCC applications presented earlier, while being as concise as possible.

The core entities in the abstraction are presented in Table 1.

Table 1: Microtasks Nomenclature

Author	A person or organization who builds and deploys a Microtasks application.
Actor	A user who interacts with a Task.
Task	Data representing a unit of work, composed of Task Input, Task Constraints and a list of submitted Solutions
Task Input	Initial Task data submitted by the Author

Activity	Code that enables a specific kind of interaction between actors and tasks. Contains **Logic** and the **UserInteraction** components.
Solution	Data appended to a Task by one or more Actors participating in an Activity.
Constraints	Metadata used to match Tasks, Actors and Activities
Project	Top level container used for establishing scope. Contains Activities, and Tasks
Room	A nexus of collaboration, consisting of an Activity, a Task list and an Actor list.

The relationships among the entities are illustrated in Figure 2. Note that this is a logical model (the implementation can run on multiple machines).

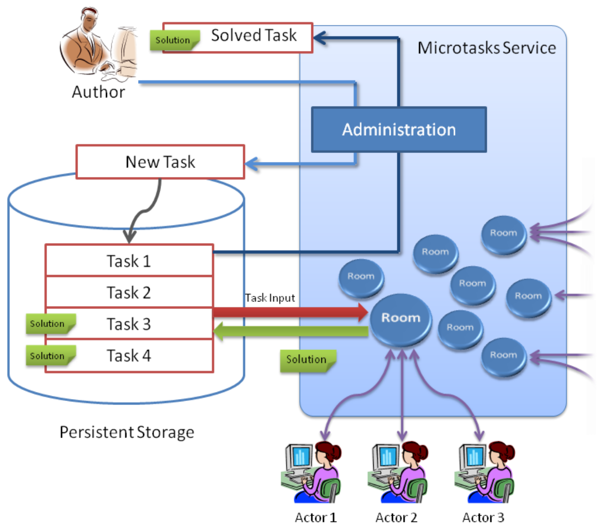


Figure 2: The Microtasks Abstraction

The Author models the data required for the work that needs to be done as a set of Tasks, which are uploaded using an Author administration interface into persistent storage maintained by the environment. Each distinct type of user interaction is called an Activity. The **Microtasks Service** in the figure contains the runtime state of the system.

Actors work (or more generally, interact) with Tasks, appending **Solutions** to the tasks they work on. To facilitate this, the environment dynamically spawns a Microtasks abstraction called a **Room**. At any point the Actors in a Room interact with a single Task. The Room is a fundamental abstraction, as it enables a simple programming model for collaboration on a particular Activity by a set of Actors on a set of tasks. This is a unified model for single- and multi-user Activities.

Actors in a Room collaborate by exchanging messages with an Author provided, Activity-specific, **Logic** component that runs within the context of a Room. The Logic component may also maintain its own transient state. We provide more details about the internal structure of a Room later in this section. Solutions are persisted (appended to the Task record in the persistent database maintained by the environment) as they complete.

Tasks can flow through multiple Activities, where each Activity represents a specific kind of interaction between a Task and one or more Actors. All solutions are appended to the Task. The flow of Tasks through Activities is explained in more detail in section *"Life Cycle of a Task"*.

We shall illustrate these concepts using the SMA application from the Introduction. Recall that with SMA, the overall objective is to annotate and segment media clips. Each Task represents a video or audio clip. SMA has two Activities, an "*annotate*" activity and a "*review*" activity. In the *annotate* Activity, two or more Actors collaborate synchronously, using the message dispatch facilities in a room, to create annotations. These annotations are accumulated as Solutions. The *review* Activity enables an Actor to approve the quality of a particular set of annotations. This decision is also appended to the Task as a solution.

Tasks, Activities and Actors have a set of properties (name value pairs), and constraints on these properties. Constraints express compatibility requirements amongst Tasks, Actors and Activities. A Room instantiator, described later, is responsible for dynamically matching up all the Actors, Tasks and Activities in a project, spawning a Room whenever there is a critical mass of compatible Actors and Tasks for a particular Activity.

To design an application using Microtasks, the author needs to break it down into two components: the *User Interaction* (UI) component and the *Logic Engine* (LE) component. The Actors interact with the UI and their actions are encapsulated as messages and sent to the LE by messaging services provided by Microtasks. Conversely, the LE can generate output messages, which are sent to the UI. All interactions are scoped by a particular Room.

NOTE: The programming model presented in this paper, while already providing a greatly simplified and structured way to implement TCC applications, does not preclude even simpler higher-level declarative or template-based ways to program TCC applications. In fact, we expect that there is a lot of value in developing a simple design surface for TCC applications which, along with a stable of UI modules or "gadgets", will enable even non-programmers to author full-fledged TCC applications! Such a design surface would generate or synthesize applications based on the programming model described in this paper. This is a highly promising area for future work.

Lifecycle of a Task

Conceptually, the life cycle is similar to that of an assembly line as illustrated in Figure 3.

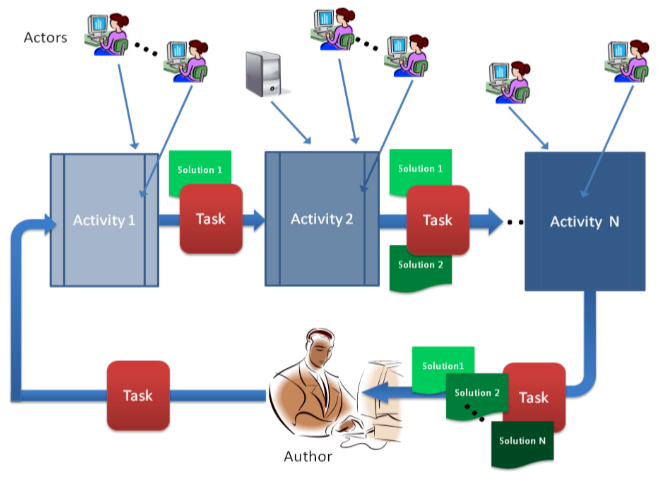


Figure 3: Lifecycle of a Task

Once created, a Task is placed on the assembly line that circulates it through various stages defined for working on it. Each stage constitutes an Activity that is performed on the Task by one or more Actors, who add solutions to it. These solutions are also added to the state of the Task and can be accessed during the subsequent stages. In any particular stage, the Task is the combination of the original Task, and the solutions added to it by the activities preceding the current Activity. More formally, T_n, the state of Task T at the start of the n^{th} stage to have operated on T, is given by the recurrence:

$$T_n = T_{n-1} \cup S_{n-1}$$
$$T_0 = \{I, C\}$$

where I is the Task Input, C is the Task constraints, and S_k is a solution added to the Task in stage k. For example, state of the Task in the "review" Activity of SMA consists of the video clip and the annotations added in the preceding "annotate" Activity.

Authors can access tasks and solutions at any stage in the activity chain, and introduce new activities and tasks dynamically.

Rooms

The Microtasks Room scopes all interactions concerning a specific set of Actors bound to an Activity and working on a set of Tasks. The mechanism for room instantiation is described later. Figure 4 shows the conceptual structure of a room.

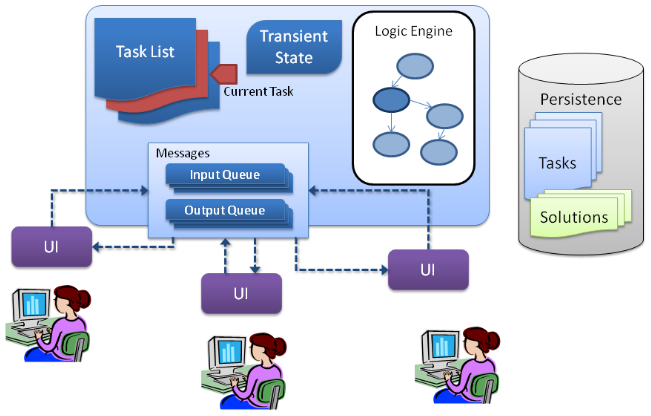


Figure 4: A Room

The "active" part of the Room is the Author-provided Logic Engine, which is invoked periodically by the Microtasks environment. Each Room has an input and output message queue that facilitate message passing between the user interaction component and the logic engine component. Current program state (for instance, variables used by the code running in the logic engine) is stored in the Room as *Transient state* which is isolated from other Rooms of the same Activity. The Logic Engine performs actions in response to messages sent by the Actors in the room. These actions include modifying the transient state, sending messages to the user interaction components, and submitting solutions to the environment.

Rooms interact with the Microtasks Environment through a set of status codes which are used to drive a finite state machine. This interaction model is described in more detail in the next section.

The Microtasks Environment manages the transmission of messages, and thus Author code does not have to concern itself with binding and transport mechanisms. Many rooms are in execution simultaneously and each of them is provided a completely isolated environment. The system manages all these Rooms, their states and the queues for messages. The system infrastructure is also responsible for a scalable implementation and all this happens in a concurrent fashion, with thread and lock management being done by the system. Authors are only expected to write simple sequential code as we describe in the "*Programming with Microtasks*" section. The model is identical for single- and multi-user activities and greatly simplifies the process of developing applications.

Dynamic Room Instantiation Based on Constraints

The Room instantiation mechanism is designed to enable flexible, constraint-based rules for matching up Actors, Activities and Tasks.

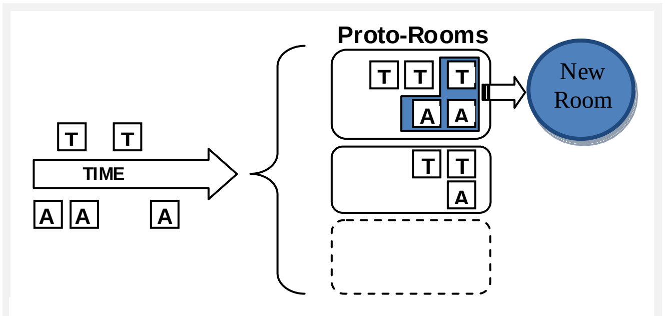


Figure 5: Dynamic Room Instantiation

A Room is instantiated for an Activity whenever there is a critical mass of mutually compatible Actors and Tasks for the Activity. Compatibility is computed from Activity constraints that can refer to fields within both Actor qualifications and Task state. For single user activities, Rooms are instantiated whenever a new Actor arrives and there are Tasks compatible with that Actor. Rooms for multi-user activities are instantiated after the minimum number of actors can be matched to a Task. Additional constraints can be added. For example, in the *annotate* Activity of the SMA application, Actors who share a common language can be paired up with specific kinds of video content, and a minimum number of Actors to actively work on a particular video can also be specified through appropriate Constraints.

Figure 5 illustrates how Rooms are instantiated, given an input stream of Actors and Tasks. This mechanism is designed for responsive instantiation of Rooms given a constant influx of Actors and Tasks. The instantiator maintains a dynamic list of "Proto Rooms", containing collections of compatible Tasks and Actors for a particular Activity. Each time an Actor *a* becomes available for an Activity, or an Activity on a Task *t* is completed, they are added to all the Proto Rooms that they are compatible with. If no Proto Rooms are available, a new one is instantiated and *a* (or *t*), is added to it. When there are enough Actors and Tasks in a Proto Room to form a Room, a Room is instantiated (and the Actors and Tasks are removed from any other Proto Rooms they had registered with).

Multistage Task Orchestration

Multi-stage Task orchestration is *emergent* behavior that stems from the way new Rooms are instantiated; one can control the Activities that successively work on a Task by setting up appropriate Activity constraints. This is best illustrated by an example. The constraint for the 2nd (*review*) stage of the SMA application is that the Task must have a solution added by the earlier *annotate* stage. We have found this constraint-based, data-driven orchestration to be flexible and simple to program, after considering other alternatives, such as defining and executing an orchestration graph.

PROGRAMMING WITH MICROTASKS

The Microtasks programming model presented in this section focuses on core concepts and is simplified for the sake of clarity of exposition.

Authors follow this application development process:

1. Define data structures to represent Task and Solution data for each Activity,
2. For each Activity:
   a. Partition the application into Logic Engine(LE) and User Interaction(UI) components
   b. Define data structures and messages for communication
   c. Code up the LE and the UI
3. Deploy the application and add tasks

We will illustrate these with the example of the SMA application.

Declaring Types

The first step in creating a Microtasks application is to define types used by the application. These include data structures for the *Task Input*, for the *Transient State*, for the *Solution*, and for the *Message*. The data structures can be arbitrarily complex, with the only constraint being that they be serializable.

For example, in the SMA application, the Task Input contains the URL for the video and the Solution for the *annotate* activity is a list of annotations for the video. The Transient State is the running list of annotations for the video. Messages are of five types – three of them define the actions that can be performed on the annotations – creation, deletion, and modification, one of them indicates that the message is a chat message and one more is reserved for indicating that the task is complete. For the *review* activity, the Task Input is the URL and the annotations added in the *annotate* activity. The Solution is the set of annotations from the Task Input, along with flags added by Actors indicating whether an annotation is useful. The message for this Activity is an annotation, flag pair, with the flag indicating if the annotation is useful.

Programming the Logic Engine

The Logic Engine(LE) code executes within the context of a Room, and is essentially a finite state machine which advances each time a message is received or optionally at periodic intervals. User actions are delivered to the Logic Engine as messages from the UI via the Room. Based on the messages, the LE can update its state and communicate back with messages to the UI. The UI will update itself based on the messages sent by the LE. For instance, the LE can send a message indicating that a session is concluded, and the UI will show the status accordingly. During each invocation of the LE, the Microtasks Environment passes to it a Context object that contains the input and output queues of the Room, and the transient state of the Activity. The LE can read the input queue, interpret the messages and take actions based on the messages received. This sequence is shown in Figure 6. The LE may also send messages back to the user interaction component, update the transient state, check the quality of solutions, and add solutions to the persistent store.

Messages between UI and LE include both event-handling messages that request some action from the logic engine (for instance, `SkipCurrentTaskMessage`), and data messages (like the `SolutionAddedMessage`).

The LE communicates with the Microtasks Environment by returning status codes that drive a state machine, which has four states: *INIT*, *RUNNING*, *COMPLETED* and *ERROR*. In the *INIT* state, the task list of the Room is populated and a current task selected, and then the state is set to *RUNNING*. After this, the LE controls the state of the Room it is running in. This is done by returning one of the appropriate symbols: *RUNNING*, *COMPLETED*, or *ERROR* after each invocation. The symbol, *RUNNING* must be returned by the LE when it determines that the current Activity isn't complete and additional invocations of the Logic Engine are required. If an LE returns the *COMPLETED* or *ERROR* symbols, the Microtasks environment will terminate the current Activity and return the Actor(s) and Task to the Room instantiator.

```
Procedure ExecuteLogicEngine
Input: Context c
Output: GameRoomStatusSymbol
begin
   state = c.getTransientState()
   messages = c.getInputMessages()
   outputs = new list
   statusSymbol = RUNNING;
   …
   <modify state and populate outputs>
   …
   c.enqueueMessagesforUsers(outputs);
   c.updateTransientState(state);
   return statusSymbol;
end
```

Figure 6: Pseudo-code for the logic engine

Programming the User Interaction

The User Interaction component interacts with the Logic Engine through messages, and is closely tied to the Logic Engine in terms of the messages it consumes and generates. The component itself can be embedded in any environment that can exchange messages with the Microtasks services, for example, in a website, mobile client or desktop application.

```
Procedure SetupUI
begin
    registerUserForActivity(activity);
    <wait till game room started>
    info = getRoomInfo();
     <setup UI and show the task>
end

Procedure ProcessUserInput
begin
    <interpret User input and create msg>
    sendMessageToLogicEngine(msg);
    r = retrieveResponseFromLogicEngine();
    <process response and update UI>
begin
```

Figure 7: Pseudo-code for the user interaction component

The pseudo-code for a typical UI component is given in Figure 7. At a high level, the user interaction component consists of two steps: First, it sets up the user interface for the Actor to work on the task data. This involves registering the actor, waiting for the Actor to be assigned to a Room, and the actual setup of the user interface using information retrieved from the Room. The information obtained includes task data, other actors in the room, and metadata like task constraints, activity constraints and Actor qualifications.

In the second step, the component responds to actor actions and sends messages (solutions, event handling messages) to the LE. Results from the LE (successful solution submission, or solutions failing validation checks) are retrieved as messages and appropriate actions taken.

The SMA application's user interaction component is a rich client front-end as shown in Figure 1. First the UI is setup, and then the video starts playing when the room starts. User actions (add/modify/delete) are intercepted by the client and encapsulated into a message and sent to the LE. The UI is updated based on messages from the LE – e.g., in case of a chat message, it is shown in the chat box, and the UI indicates completion on receiving the `TaskComplete` message.

Deploying and administering the application

The application can be compiled and the resulting library can be submitted to the Microtasks service for deployment. The administration interface is used to add tasks to the project and retrieve solutions. For instance, in case of the SMA application, we would need to submit URLs for each of the videos that are to be annotated. The user interface can be hosted on a web server (also provided by the Microtasks Service) or distributed as desktop applications.

Discussion

Our programming model is designed to allow simplicity and extensibility while developing the applications. Application logic can be written as though a single instance of the application is running. Application state can therefore be stored as part of the logic, and can be accessed without the fear of race conditions. Programming the logic engine is therefore simpler. Since the author can plug-in her own data structures and custom logic, the system is very extensible, and a diverse group of activities can be developed.

Since the application logic runs in the same environment during both development and deployment, applications can be developed and tested completely on development machines before being deployed on hosting servers, without the fear of unexpected bugs caused by differing environments. Issues of scaling the application to large numbers of instances are now handled by the platform. Isolating parallel instances of an application, as well as instances of other applications from one another is also managed by the platform. Finally, security is tightened because the logic server can be configured to disallow certain API calls, by either loading the application logic in custom runtimes, or by configuring the runtime to disallow certain API calls. This prevents hacked applications from running amok.

ARCHITECTURE, DESIGN AND IMPLEMENTATION

In this section, we delve deeper into the architecture of some of the components of the Microtasks system. The Microtasks system is designed with helper libraries and various services that interact with each other to provide functionalities to the user. The system is implemented in C# using the .NET platform and technologies such as Windows Communication Foundation and Language Integrated Query.

Architecture

The overall architecture is described in Figure 8. The Author and the Actors communicate to the Microtasks system through the *Microtasks Service* layer which provides a single *Facade* design pattern for interacting with the system. The service can be accessed over multiple transports, such as HTTP web services and .NET Remoting. This ensures flexibility and allows both browser-based and desktop clients to connect to it.

The *Persistence* layer abstracts database storage and query, and is used internally by all the layers above it, including the management of Author data. It also checks data for consistency before adding it to the database.

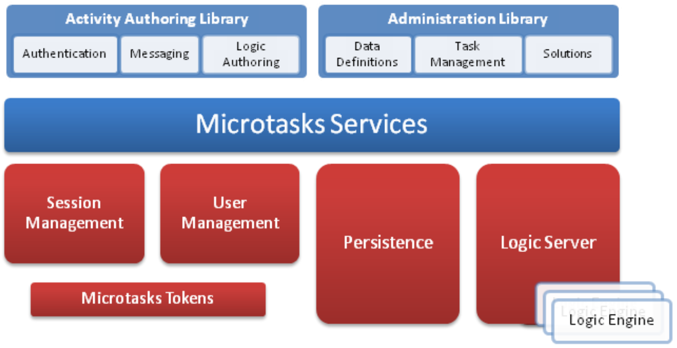


Figure 8: Microtasks Architecture

The *Session Management* Service is responsible for access control and unified session management for both desktop and browser-based applications. It supports multiple authentication mechanisms. Authors can also choose to allow anonymous access to certain Activities. Timed sessions are assigned to users after authentication for a specific role (Author or different Actor roles). All entities in the Microtasks Environment (Sessions, Authors, Actors, Activities and Tasks) are allocated a Token, which encapsulates their globally-unique id. Tokens contain discriminators for identifying the type of parent entity and are used at various levels to check for access privileges based on user role. Tokens are also used for tracking relationships between entities, as part of a monitoring framework.

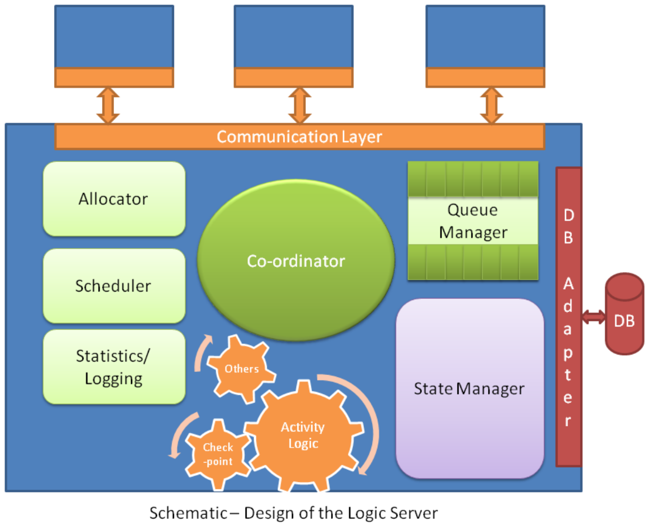


Figure 9: The Logic Server Architecture

Once the session is created, the *Logic Server* (Figure 9) manages the actual activities being executed. This is central to the system and manages user-allocation, Rooms, message queues and the scheduling of the various Logic Engines. The various components of the Logic Server are listed below:

- **Allocator:** Enables matching of Actors with Tasks, based on the constraints as described in "*Dynamic Room Instantiation Based on Constraints.*"
- **Queue Manager:** Maintains message queues for actors and rooms. Message queues for Actors get cleared when the Actor retrieves messages, and for rooms when the context is set up.
- **State Manager:** The state manager behaves as a store that maintains the transient state of all the activities in execution. When a room is created, the state is initialized to the default state provided by the Author, and thereon, the state is retrieved from the store and handed to the Logic Engine during execution and deposited back when execution completes.
- **Finite State Machine (FSM) Manager:** As we described in the programming model, the state of the rooms (Init, Running, Exit and Checkpoint) is maintained as a finite state machine. The FSM Manager finds state transitions, and also schedules bookkeeping activities like cleanup, check-pointing etc. This design allows us to extend the control-state management easily by modifying the automaton.
- **Logic Engine Executor:** This component executes the logic engines provided by the Authors. Before running the Activity Logic Engine (LE), the Executor prepares a context populating it with the currently queued messages, the transient state, and information about the room.

The Queue-Manager and the State-Manager need to maintain states for all the players under the protection of proper locks. The Logic Server is designed to be highly concurrent and executes many LEs simultaneously in different threads. The **Scheduler** manages the thread pools and queues up the Logic Engines to be executed as per the specified *tick-time*, and the availability of the worker threads. The **Co-ordinator** ties all these modules together. Note that this infrastructure is common to all the games and can be well-engineered. All the multi-threading and lock management complexity is hidden within this layer so that the Author of the application can write sequential code.

The Author's logic is run within the context of the Microtasks Service allowing the service to ensure that the environment is sand-boxed and failure of Author's code does not affect the rest of the system. Since the operational semantics of the LE are restricted to message passing and does not need to store private transient state, static verification that the LE does not contain malicious code is viable.

EXPERIENCE WITH MICROTASKS APPLICATIONS

In this section we describe the benefits of developing applications with the concepts and environment described earlier in the paper. An implementation of the Microtasks environment is currently deployed in our organization and is being used for the development of applications internally.

Our team has developed several applications motivated by real scenarios for annotation, data collection and collaborative tasks. The **Image Annotation** application collects training and verification data for computer vision systems. The **Multilingual Map (MLMap)** Annotation project, described later, was developed to collect transliteration data for improving multi-lingual map search, and the **SMA Shared Media Annotation** project, to help annotate educational videos.

Table 2: Application Code Sizes (LOC)

App	Image Annotation	SMA	ML Map	Stress
UI	609	398	310+	70
Data Structures	93	50	30	48
Logic Engine and Messages	93	160	68	45
Admin	210	26	151	22

Through these examples of applications built using Microtasks we aim to draw attention to the minimal code size required to implement various components. As mentioned in Table 2, this is true both for applications designed to be used by Actors individually as well as those which require synchronous collaboration. Based on our experience, comparable featured applications would easily be in excess of 4000 LOC, and far more complex in structure.

We now qualitatively describe our experiences with these applications.

Multilingual Map Transliteration

The *Multilingual Map Transliteration* application has been developed to get human transliterations of place names in various languages. The Activity, as shown in Figure 10, is deployed as a web-based mashup. The activity accesses various online services to render the transliteration task (that is the name of a specific place) on the map and display images from nearby areas.

The use of Microtasks allowed us to incrementally add features to the user interaction and Logic Engine components, without losing out on already existing solutions submitted by Actors. We have experimented with more than a couple of user interaction components that interact with the same Logic Engine without the need for investing in the design and development of our own web services. This was possible as the Microtasks server is easily accessible over HTTP web services. Since the Logic Engine of any particular activity can be updated easily we have been able to add new features such as a transliteration engine that pre-populates the Task with a machine transliteration that an Actor can correct.

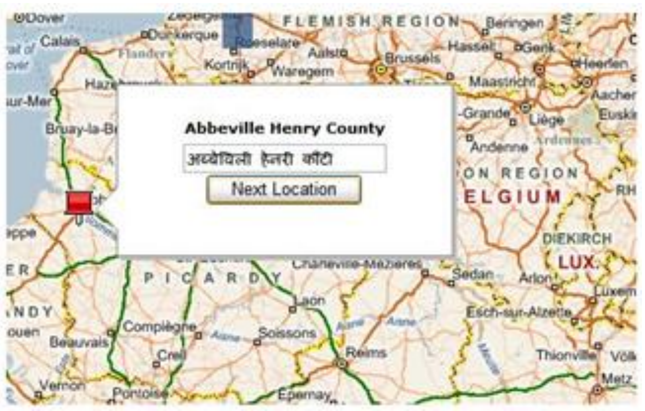


Figure 10: Multilingual Map Transliteration Website

Image Annotation

The availability of appropriately tagged and annotated images is an important requirement for most machine learning based computer vision approaches. Researchers require annotated data for both training and testing of their algorithms, many times on very specific image sets.

Our implementation of the Image Annotation Project contains one browser-based Activity, shown in Figure 11, which allows an Actor to annotate specific parts of the image with text. This *single-user* activity is developed using the Microsoft Silverlight platform and hosted in the enterprise domain. On visiting the website, the Actor is automatically authenticated and assigned an untagged image. Also, the Author can review annotations submitted by Actors at any time using the reviewing application.

Session management and authentication for participating Actors is handled by the Microtasks client libraries and the Session Services layer and greatly simplifies the development of the web client. Building this web application using Microtasks also eliminated the need for any database setup or persistence related issues, further helping us to rapidly prototype and experiment with the client interaction components.

Shared Media Annotation (SMA)

The Shared Media Annotation application is the example we have been using throughout our paper. It is a viable option for the annotation and segmentation of educational videos to support educators [21] and also improve search over a media repository. Annotations created by the actors in the same Room are shared live. This particular project strongly demonstrates the effort saving benefits of the environment as the entire application required less than 40 person-hours for completion, and is now a fully functional application in active use. In this short time, we have been able to achieve many of the design requirements for Collaborative Video Annotation tools designed from scratch such as FilmEd[22] which involved significant effort for design and development.

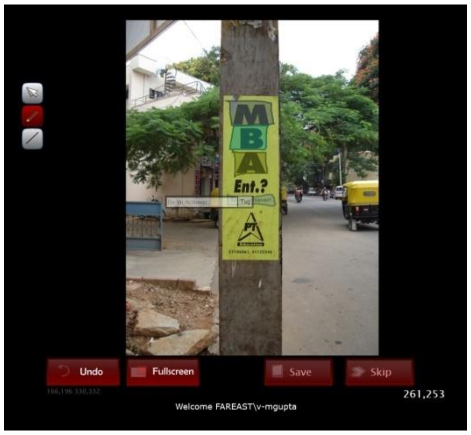


Figure 11: The Image Annotation Activity

The application was initially developed and tested as a single user application, and was seamlessly extended to work with multiple users working on a Task synchronously. The Logic Engine for SMA is completely linear as the Logic Server isolates Rooms and also handles message queuing. As part of the Logic Engine, traces of user activity are extremely trivial to persist (as Messages) and we are hoping to leverage the same to understand the limits and dynamics of collaborative annotation.

Using the framework has another advantage in increasing security. Work done at the USC Center for Software Engineering [23] has estimated a lower bound increase of 30% in software construction costs, and 24% in elaboration (design) costs for writing **secure** applications. We believe a significant percentage of these costs are reduced while deploying applications using the Microtasks environment, as a large number of security features like session management, authentication and access control are provided by it.

In practice, Microtasks reduces effort in analysis, design and development. Development and testing effort is reduced both by the functionality offered and by the programming model. The Logic Server enables developers to write applications with very little concern for concurrency issues, nearly eliminating a large class of (synchronization-related) bugs. Performance concerns, particularly the ability of the infrastructure to handle large loads are now a concern of the framework developers and are no longer a burden on the application developer. Finally, testing effort is reduced because the platform supports a what-you-debug-is-what-you-deploy environment in which business logic runs in the same environment during development and deployment.

CONCLUSION

We have formulated the category of TCC applications, called out common features of such applications, and provided an abstraction for specifying, designing and developing them. The abstraction identifies common services used by this class of applications, abstracts them into a framework, and in addition provides a programming model that is well-suited for the development of these applications. Our implementation of the abstraction has allowed us to significantly lower the barrier for developing these applications and develop and deploy many real and diverse applications for internal organizational use. The ability to track and monitor actor behavior with the ability to rapidly make changes to these applications are helping us better understand the nuances of collaborative application design and user motivation. Our future work includes building a design surface with back-end code generation that would enable even non-programmers to build and deploy TCC applications.

REFERENCES

1. Distributed Proofreaders. *Project Gutenberg.* [Online] http://www.pgdp.net/.

2. *Project Gutenberg.* [Online] http://www.gutenberg.org/wiki/Main_Page.

3. *Labeling Images with a Computer Game.* **Ahn, Luis von and Dabbish, Laura.** Vienna, Austria : ACM, 2004. Conference on Human Factors in Computing Systems. pp. 319 - 326. ISBN:1-58113-702-8.

4. **Barr, Jeff and Caberera, Luis Felipe.** AI Gets a Brain. *Queue.* May 2006, Vol. 4, 4, pp. 24 - 29.

5. Microsoft Task Market. [Online] http://www.taskmarket.com.

6. **Ahn, Luis von.** Games with a purpose. *Games with a purpose.* [Online] http://www.gwap.com/gwap/.

7. —. Games with a Purpose. *IEEE Computer Magazine.* June 2006, pp. pp 96-98.

8. **Douglis, Fred.** From the Editor in Chief: The Search for Jim, and the Search for Altruism. *Internet Computing, IEEE.* May-June, 2007, Vol. 11, 3.

9. **Julian Birkinshaw, Stuart Crainer.** Game on: Theory Y meets Generation Y. *Business Strategy Review.* Winter 2008, pp. 4-10.

10. Dolores Labs. [Online] http://doloreslabs.com.

11. Seriosity. [Online] http://www.seriosity.com.

12. **Howe, Jeff.** The Rise of Crowdsourcing. *Wired.* June 2006.

13. *Character recognition in natural images.* **T. E. de Campos, B. R. Babu, and M.Varma.** Lisbon, Portugal : s.n., February 2009. International Conference on Computer Vision Theory and Applications.

14. *Hot Or Not.* [Online] http://www.hotornot.com.

15. *Flickr.* [Online] http://www.flickr.com/.

16. *Peekaboom: A Game for Locating Objects in Images.* **Ahn, Luis von, Liu, Ruoran and Blum, Manuel.** Montréal, Québec, Canada : ACM, 2006. Conference on Human Factors in Computing Systems. pp. 55 - 64.

17. **Phillips, William Greg.** *Architectures for Synchronous Groupware.* Department of Computing and Information Science, Queen's University. Kingston, Ontario : s.n., 1999. Technical Report. 0836-0227-1999-425.

18. Yotta DCL. *Yotta Geo Spatial Vision.* [Online] http://www.yotta.tv/contact/.

19. **Sweeney, Tim.** Unreal Networking Architecture. [Online] August 21, 1999. http://unreal.epicgames.com/Network.htm.

20. *Entering the Education Arcade.* **Jenkins, Henry, et al.** 1, s.l. : ACM, 2003, Vol. 1. 1544-3574.

21. **Team, DSH.** *The Digital StudyHall.* s.l. : Department of Computer Science and Engineering, University of Washington, August 2007. Technical Report UW-CSE-07-08-01.

22. *FilmEd - collaborative video indexing, annotation, and discussion tools over broadband networks.* **Schroeter, R, Hunter, J. and Kosovic, D.** s.l. : IEEE, 2004, Vols. Multimedia Modelling Conference, 2004. Proceedings. 10th International. 0-7695-2084-7.

23. **Edward Colbert, Murali Gangadharan, Donald Reifer, and Barry Boehm.** Extending COCOMO II to estimate the cost of developing secure software. *USC Center for Software Engineering.* [Online] http://sunset.usc.edu/GSAW/gsaw2003/s5/colbert.pdf.